CLAIMS

1. A method for designing an application programming interface (API), the method comprising:

preparing a plurality of code samples for a core scenario, each respective code sample of the plurality of code samples corresponding to a respective programming language of a plurality of programming languages; and

deriving the API from the core scenario responsive to the plurality of code samples.

2. The method as recited in claim 1, further comprising:

determining, by an API designer, if the derived API is too complex.

3. The method as recited in claim 2, further comprising:

if the derived API is determined to be too complex, then

refining, by the API designer, the derived API to produce a refined API.

4. The method as recited in claim 3, further comprising:

determining, by the API designer, if the refined API is too complex.

5. The method as recited in claim 1, further comprising:

performing one or more usability studies on the API utilizing a plurality of developers.

6. The method as recited in claim 5, wherein the performing comprises:

performing the one or more usability studies on the API utilizing the plurality of developers wherein the plurality of developers are typical for the plurality of programming languages.

7. The method as recited in claim 5, further comprising:

ascertaining whether the plurality of developers are able to use the API without significant problems.

8. The method as recited in claim 7, further comprising:

if the plurality of developers are not ascertained to be able to use the API without significant problems, then revising the API.

9. The method as recited in claim 8, wherein the revising comprises:

revising the API based on at least one lesson from the one or more usability studies.

10. The method as recited in claim 1, wherein the deriving comprises:

deriving the API to support the plurality of code samples that correspond respectively to the plurality of programming languages.

11. The method as recited in claim 1, wherein the deriving comprises:

gleaning language-specific mandates from the plurality of code samples; and

incorporating the language-specific mandates into the API.

12. The method as recited in claim 1, wherein the deriving comprises:

gleaning language-inspired developer expectations from the plurality of code samples; and

incorporating the language-inspired developer expectations into the API.

13. The method as recited in claim 1, wherein the deriving comprises:

gleaning commonalities from the plurality of code samples; and

incorporating the commonalities into the API.

14. The method as recited in claim 1, wherein the deriving comprises:

deriving the API to have an aggregate component that ties a plurality of lower-level factored types together to support the core scenario.

15. A method for designing an application programming interface (API), the method comprising:

selecting a core scenario for a feature area;

writing at least one code sample for the core scenario; and

deriving an API for the core scenario responsive to the at least one code sample.

16. The method as recited in claim 15, wherein the selecting comprises:

selecting a plurality of core scenarios for the feature area.

17. The method as recited in claim 16, further comprising:

repeating the writing and the deriving for each core scenario of the plurality of core scenarios that are selected for the feature area.

18. The method as recited in claim 15, wherein the writing comprises:

writing a plurality of code samples for the core scenario, each respective code sample of the plurality of code samples corresponding to a respective programming language of a plurality of programming languages.

19. The method as recited in claim 18, wherein the deriving comprises:

deriving the API for the core scenario responsive to the plurality of code samples.

20. The method as recited in claim 15, further comprising:

performing one or more usability studies on the API utilizing a plurality of developers;

ascertaining whether the plurality of developers are able to use the API without significant problems; and

if the plurality of developers are not ascertained to be able to use the API without significant problems, then revising the API.

21. The method as recited in claim 20, wherein the revising comprises:

revising the API based on at least one lesson from the one or more usability studies to produce a revised API.

22. The method as recited in claim 21, further comprising:

repeating the performing and the ascertaining with respect to the revised API.

23. The method as recited in claim 15, wherein the deriving comprises:

deriving the API to support the at least one code sample written for the core scenario by producing a two-layer API that includes an aggregate component and a plurality of underlying factored types.

24. The method as recited in claim 15, wherein the deriving comprises:

gleaning one or more language-specific mandates from the at least one code sample; and

incorporating the one or more language-specific mandates into the API.

25. The method as recited in claim 15, wherein the deriving comprises:

encapsulating a particular factored type into an aggregate component that is associated with the core scenario if all members of the particular factored type are exposed by the aggregate component.

26. The method as recited in claim 15, wherein the deriving comprises:

encapsulating a particular factored type into an aggregate component that is associated with the core scenario if the particular factored type is independently uninteresting to other component types.

27. The method as recited in claim 15, wherein the deriving comprises:

exposing a particular factored type from an aggregate component that is associated with the core scenario if at least one member of the particular factored type is not exposed by the aggregate component.

28. The method as recited in claim 15, wherein the deriving comprises:

exposing a particular factored type from an aggregate component that is associated with the core scenario if the particular factored type can be beneficially used independently of the aggregate component by another component type.

29. The method as recited in claim 15, wherein the deriving comprises:

producing a two-layer framework that includes component types targeting a relatively higher level of abstraction and component types targeting a relatively lower level of abstraction.

30. The method as recited in claim 29, wherein the component types targeting the relatively higher level of abstraction are directed to core scenarios.

31. The method as recited in claim 29, wherein the component types targeting the relatively lower level of abstraction provide a relatively greater amount of control to developers as compared to the component types targeting the relatively higher level of abstraction.

32. The method as recited in claim 15, wherein the deriving comprises:

deriving the API so as to enable a developer to implement a create-set-call usage pattern for the core scenario.

33. The method as recited in claim 32, wherein the deriving comprises:

producing the API with pre-selected parameters that are appropriate for the core scenario.

34. A method for designing an application programming interface (API), the method comprising:

deriving an API for a scenario responsive to at least one code sample written with regard to the scenario;

performing one or more usability studies on the API utilizing a plurality of developers; and

revising the API based on the one or more usability studies.

35. The method as recited in claim 34, further comprising:

writing a plurality of code samples with regard to the scenario, each respective code sample of the plurality of code samples corresponding to a respective programming language of a plurality of programming languages;

wherein the deriving comprises:

deriving the API for the scenario responsive to the plurality of code samples.

36. The method as recited in claim 34, further comprising, prior to the performing one or more usability studies on the API:

determining, by an API designer, if the derived API is too complex;

if the derived API is determined to be too complex, then

refining, by the API designer, the derived API to produce a refined API; and

determining, by the API designer, if the refined API is too complex.

37. The method as recited in claim 34, further comprising:

ascertaining whether the plurality of developers are able to use the API without significant problems; and

when the plurality of developers are not ascertained to be able to use the API without significant problems, then implementing the revising;

wherein the revising comprises:

revising the API based on at least one lesson from the one or more usability studies to produce a revised API.

38. The method as recited in claim 37, further comprising:

repeating at least the performing and the ascertaining with respect to the revised API.

39. The method as recited in claim 37, wherein the ascertaining comprises:

ascertaining whether the plurality of developers are able to use the API without significant problems with regard to a desired level of usability for at least one targeted developer group, wherein the desired level of usability includes considerations with respect to (i) frequent and/or extensive reference to detailed API documentation by the plurality of developers, (ii) a failure of a majority of the plurality of developers to implement the scenario, and (iii) whether the plurality of developers take an approach that is significantly different from what is expected by an API designer.

40. The method as recited in claim 34, further comprising:

selecting a plurality of core scenarios for a feature area;

repeating the deriving, the performing, and the revising for each core scenario of the plurality of core scenarios.

41. The method as recited in claim 40, wherein the deriving comprises:

producing an aggregate component for each core scenario of the plurality of core scenarios.

42. The method as recited in claim 34, wherein the deriving comprises:

producing an aggregate component that has a respective relationship with each respective factored type of a plurality of factored types.

43. The method as recited in claim 42, wherein the producing comprises:

producing the aggregate component to support the scenario for which the at least one code sample is written.

44. The method as recited in claim 42, wherein the producing comprises:

producing the aggregate component to have an exposed relationship with at least one factored type of the plurality of factored types and an encapsulated relationship with at least one other factored type of the plurality of factored types.

45. The method as recited in claim 44, wherein the at least one other factored type of the plurality of factored types that has the encapsulated relationship with the aggregate component can be handed off by the aggregate component for direct interaction with another component type.

46. The method as recited in claim 42, wherein the plurality of factored types are designed using an object-oriented methodology.

47. A method for designing an application programming interface (API), the method comprising:

preparing a plurality of code samples for a core scenario, each respective code sample of the plurality of code samples corresponding to a respective programming language of a plurality of programming languages;

deriving the API for the core scenario responsive to the plurality of code samples;

performing one or more usability studies on the API utilizing a plurality of developers; and

revising the API based on the one or more usability studies.

48. A method for designing an application programming interface (API), the method comprising:

writing at least one code sample for a scenario; and

deriving an API for the scenario responsive to the at least one code sample, the API including (i) an aggregate component that is adapted to facilitate implementation of the scenario and (ii) a plurality of factored types that provide underlying functionality for the aggregate component, the API enabling a gradual progression from using the aggregate component in simpler situations to using an increasing portion of the plurality of factored types in increasingly complex situations.

49. A method for designing an application programming interface (API), the method comprising:

deriving at least one aggregate component to support at least one code sample for at least one scenario;

determining additional requirements with respect to the at least one scenario;

deciding if the additional requirements can be added to the at least one aggregate component without adding undue complexity to the at least one scenario; and

if not, defining a plurality of factored types responsive to the deciding.

50. The method as recited in claim 49, further comprising:

if so, refining the at least one aggregate component to incorporate the additional requirements.

51. The method as recited in claim 49, further comprising:

selecting a plurality of core scenarios for a feature area, the plurality of core scenarios including the at least one scenario; and

writing a plurality of code samples showing preferred lines of code for the plurality of core scenarios, the plurality of code samples including the at least one code sample;

wherein the deriving comprises:

deriving a plurality of aggregate components, which include the at least one aggregate component, to support the plurality of code samples for the plurality of core scenarios.

52. The method as recited in claim 49, wherein the deriving comprises:

deriving the at least one aggregate component with appropriate methods, defaults, and abstractions to support the at least one code sample for the at least one scenario.

53. The method as recited in claim 49, further comprising:

refining the at least one code sample according to the at least one derived aggregate component; and

evaluating the refined at least one code sample with regard to simplicity; and

repeating the deriving if the refined at least one code sample fails to be sufficiently simple in the evaluating.

54. The method as recited in claim 49, wherein the determining comprises:

determining additional requirements with respect to the at least one scenario, wherein the additional requirements include additional scenarios, additional usages, and additional interactions with other component types.

55. The method as recited in claim 49, wherein the deciding comprises:

considering whether adding the additional requirements to the at least one aggregate component hinders a create-set-call usage pattern.

56. The method as recited in claim 49, wherein the defining comprises:

defining the plurality of factored types responsive to the deciding with an ideal factoring of a full set of functionality.

57. The method as recited in claim 49, wherein the defining comprises:

defining the plurality of factored types responsive to the deciding using one or more object-oriented methodologies.

58. The method as recited in claim 49, further comprising:

determining whether the at least one aggregate component is to encapsulate or expose the functionality of each factored type of the plurality of factored types.

59. The method as recited in claim 49, further comprising:

refining the plurality of factored types to support the at least one aggregate component and the additional requirements.